Detection and Recovery

Daniel Plakosh, Software Engineering Institute [vita[1]]

2006-01-11; Updated 2008-10-06 L3 / D/P, L[2]

There are a number of runtime solutions that can detect stack corruption and buffer overruns or guard against attacks. These solutions typically terminate the program when an anomaly is detected, preventing the execution of arbitrary code.

Development Context

Program runtime checks and protection techniques that can be used to detect stack corruption and buffer overruns or guard against attacks

Technology Context

C, UNIX, WIN32

Attacks

Attacker executes arbitrary code on machine with permissions of compromised process or changes the behavior of the program.

Risk

Programming errors can result in buffer overflow vulnerabilities.

Description

Detection and recovery mitigation strategies generally make changes to the run-time environment to detect buffer overflows when they occur so that the application or operating system can recover from the error (or at least fail safely). Detection and recovery mitigations generally form a second line of defense in case the “outer perimeter” is compromised. Because attackers have numerous options for controlling execution after a buffer overflow has occurred, detection and recovery is not as effective as prevention and should not be depended on as the only mitigation strategy.

Compiler-Generated Runtime Checks

Microsoft Visual C++ provides native runtime checks to catch common runtime errors such as stack pointer corruption and overruns of local arrays at debug. Visual C++ also provides a runtime_checks pragma that disables or restores the /RTC settings.

Stack Pointer Corruption. Stack pointer verification detects stack pointer corruption. Stack pointer corruption can be caused by a calling convention mismatch.

Overruns of Local Arrays. The /RTCs option enables stack frame runtime error checking for writes outside the bounds of local variables such as arrays but does not detect overruns when accessing memory that results from compiler padding within a structure.

1. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/268-BSI.html (Plakosh, Daniel)

Nonexecutable Stacks

Nonexecutable stacks are a runtime solution to buffer overflows that are designed to prevent executable code from running in the stack segment. Many operating systems can be configured to use nonexecutable stacks.

Nonexecutable stacks are often represented as a panacea in securing against buffer overflow vulnerabilities. However, nonexecutable stacks do not prevent buffer overflows from occurring in the stack, heap, or data segments. They do not prevent an attacker from using a buffer overflow to modify a return address, variable, data pointer, or function pointer. Nonexecutable stacks do not prevent arc injection or injection of the execution code in the heap or data segments. Not allowing an attacker to run executable code on the stack can prevent the exploitation of some vulnerabilities, but it is often only a minor inconvenience to an attacker.

Stackgap

Many stack-based buffer overflow exploits rely on the buffer being at a known location in memory. If the attacker can overwrite the function return address, which is at a fixed location in the overflow buffer, execution of the attacker-supplied code starts. Introducing a randomly sized gap of space upon allocation of stack memory makes it more difficult for an attacker to locate a return address on the stack while costing no more than one page of real memory. This mitigation can be relatively easy to add to an operating system. Figure 1 shows the change to the Linux kernel required to implement Stackgap.

Although Stackgap may make it more difficult to exploit a vulnerability, it does not prevent exploits if an attacker can use relative, rather than absolute, values.

Figure 1. Linux kernel modification to support stackgap

```
1. sgap = STACKGAPLEN;
2. if (stackgap_random != 0)
      sgap += (arc4random() * ALIGNBYTES) & (stackgap_random - 1);
   /* Check if args & environ fit into new stack */
3. len = ((argc + envc + 2 + pack.ep_emul->e_arglen) *
         sizeof(char *) + sizeof(long) + dp + sgap +
         sizeof(struct ps_strings)) - argp;
```

Runtime Bounds Checkers

If using a type-safe language such as Java is impractical, it may still be possible to use a compiler that performs array bounds checking for C programs.

Jones and Kelley [Jones 97] propose an approach for bounds checking using referent objects. This approach is based on the principle that an address computed from an in-bounds pointer must share the same referent object as the original pointer. Unfortunately, there are a surprisingly large number of programs that generate and store out-of-bounds addresses and later retrieve these values in their computation without causing buffer overflows—making these programs incompatible with this bounds-checking approach. This approach to runtime bounds checking also has significant performance costs, particularly in pointer-intensive programs, where performance may slow down by up to 30 times [Cowan 00].

Ruwase and Lam have improved the Jones and Kelley approach in their C Range Error Detector (CRED) [Ruwase 04]. According to the authors, CRED enforces a relaxed standard of correctness by allowing program manipulations of out-of-bounds addresses that do not result in buffer overflows. This relaxed standard of correctness provides higher compatibility with existing software.

CRED can be configured to check all bounds of all data or of string data only. Full bounds checking, like the Jones and Kelley approach, imposes significant performance overhead. Limiting the bounds checking to strings improves the performance for most programs. Overhead ranges from 1% to 130% depending on the use of strings in the application.

Bounds checking is effective in preventing most overflow conditions but is not perfect. The CRED solution, for example, is unable to detect conditions where an out-of-bounds pointer is cast to an integer, used in an arithmetic operation, and cast back to a pointer. The approach does prevent overflows in the stack, heap, and

data segments. CRED was effective in detecting 20 different buffer overflow attacks developed by Wilander and Kamkar for evaluating dynamic buffer overflow detectors [Wilander 03], even when optimized to check only for overflows in strings.

CRED has been merged into the latest Jones and Kelly checker for GCC 3.3.1, which is currently maintained by Herman ten Brugge[29].

Canaries

Canaries are another mechanism used to detect and prevent stack smashing attacks. Instead of performing generalized bounds checking, canaries are used to protect the return address on the stack from sequential writes through memory (for example, resulting from a `strcpy()`). Canaries consist of a hard-to-insert or hard-to-spoof value written to an address below the section of the stack being protected. A sequential write would therefore need to overwrite this value on the way to the protected region. The canary is initialized immediately after the return address is saved and checked immediately before the return address is accessed.

A hard-to-insert or terminator canary consists of four different string terminators (CR, LF, NULL, and -1). This guards against buffer overflows caused by string operations but not memory copy operations.

A hard-to-spoof or random canary is a 32-bit secret random number that changes each time the program is executed. This approach works well as long as the canary remains a secret.

Canaries are implemented in StackGuard [Cowan 98]. Various StackGuard versions have been used with GCC for Immunix OS 6.2, 7.0, and 7+. Red Hat 7.3 plans to merge StackGuard 3 into the GCC 3.x mainline compiler. Canaries have also been used in ProPolice and Microsoft's Visual C++ .NET.

Canaries are useful only against exploits that overflow a buffer on the stack and attempt to overwrite the stack pointer or other protected region. Canaries do not protect against exploits that modify variables, data pointers, or function pointers. Canaries do not prevent buffer overflows from occurring in any location, including the stack segment.

Neither the terminator or random canary offers complete protection against exploits that overwrite the return address. Exploits that write four bytes directly to the location of the return address on the stack can defeat terminator and random canaries [Bulba 00]. To solve these direct access exploits, StackGuard added Random XOR canaries [Wagle 03] that XOR the return address with the canary. Again, this works well as long as the canary remains a secret.

Stack Smashing Protector (ProPolice)

A popular mitigation approach derived from StackGuard is the GCC Stack Smashing Protector (SSP, also known as ProPolice) [Etoh 00]. SSP is a GCC extension for protecting applications written in C from the most common forms of stack buffer overflow exploits and is implemented as an intermediate language translator of GCC. SSP provides buffer overflow detection and the variable reordering to avoid the corruption of pointers. Specifically, SSP reorders local variables to place buffers after pointers and copies pointers in function arguments to an area preceding local variable buffers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations.

The SSP feature is enabled using gcc options. The -fstack-protector and -fno-stack-protector options enable and disable stack smashing protection. The -fstack-protector-all and -fno-stack-protector-all options enable and disable the protection of every function, not just the functions with character arrays.

SSP works by introducing a guard variable to prevent changes to the arguments, return address, and previous frame pointer. Given the source code of a function, a preprocessing step inserts code fragments into appropriate locations as follows:

- Declaration of local variables

29. http://web.inter.nl.net/hcc/Haj.Ten.Brugge

```
volatile int guard;
```

- Entry point

```
guard = guard_value;
```

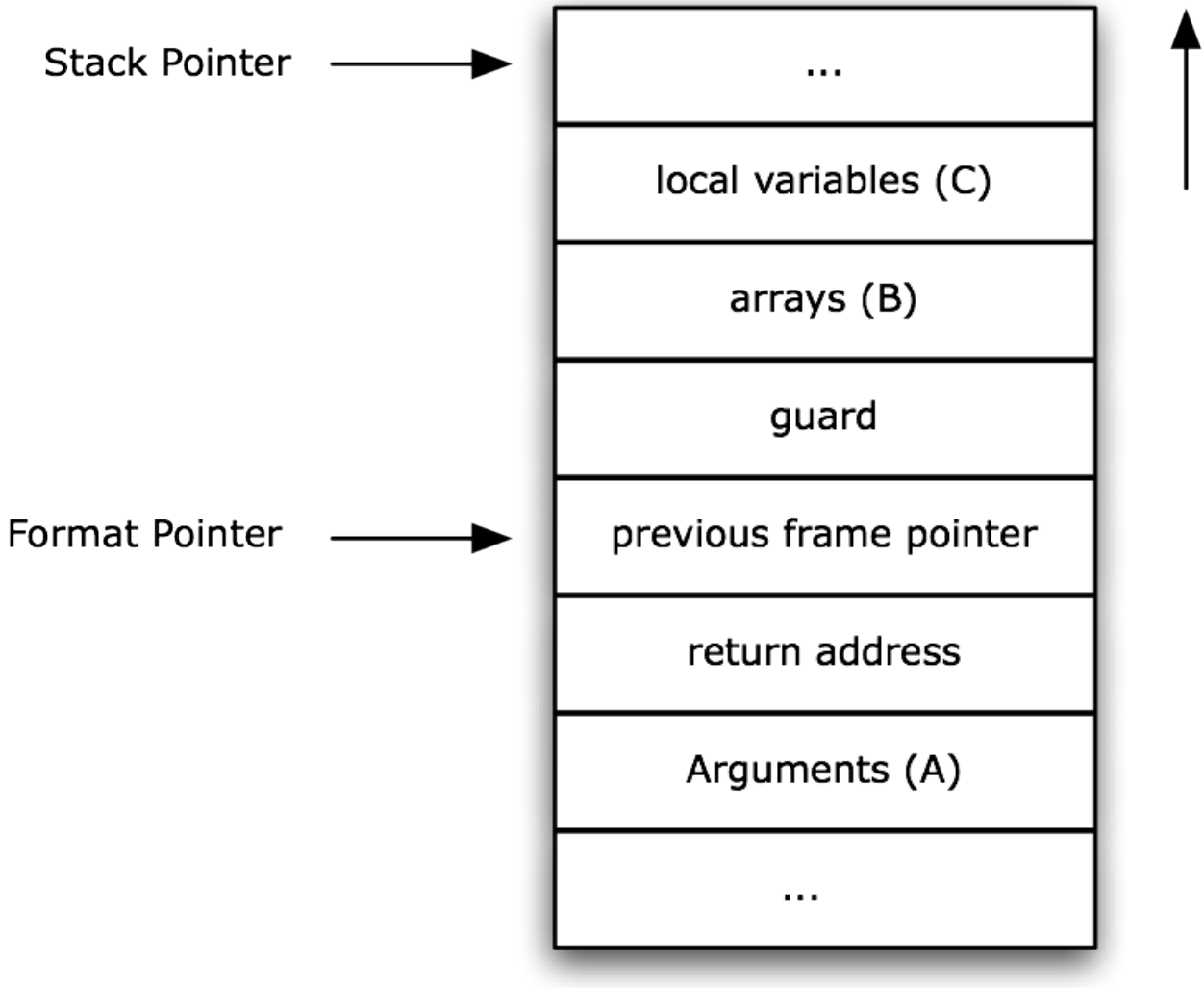
- Exit point

```
if (guard != guard_value) {
/* output error log */
/* halt execution */
}
```

A random number is generated for the guard value during application initialization, preventing discovery by a nonprivileged user.

SSP also provides a safer stack structure, as shown in Figure 2.

Figure 2. SSP safe frame structure



This structure establishes the following constraints:

- Location (A) has no array or pointer variables.
- Location (B) has arrays or structures that contain arrays.
- Location (C) has no arrays.

Placing the guard after the section containing the arrays (B) prevents a buffer overflow from overwriting the arguments, return address, previous frame pointer, or local variables (but not other arrays).

Libsafe and Libverify

Libsafe is a dynamic library available from Avaya Labs Research[57] for limiting the impact of buffer overflows on the stack. The library intercepts and bounds checks arguments to C library functions that are susceptible to buffer overflow [Baratloo 00]. The library makes sure that frame pointers and return addresses cannot be overwritten by an intercepted function. The Libverify library, also described by Baratloo et al. [Baratloo 00], implements a return address verification scheme similar to that used in StackGuard but does not require recompilation of source code, which allows it to be used with existing binaries.

Summary

Runtime solutions such as bounds checkers, canaries, and safe libraries also have a runtime performance cost and sometimes compete with each other. For example, it may not make sense to use a canary in conjunction with safe libraries because each performs more or less the same function in a different way.

References

[Baratloo 00] Baratloo, A.; Singh, N.; & Tsai, T. “Transparent Run-Time Defense Against Stack Smashing Attacks,” 251-262. *Proceedings of 2000 USENIX Annual Technical Conference*. San Diego, CA, June 18-23, 2000. Berkeley, CA: USENIX Association, 2000.

[Bulba 00] Bulba & Kil3r. *Bypassing StackGuard and StackShiel* (Prack, Volume 0xa Issue 0x38 05.01.2000 0x05[0x10]). http://www.phrack.org/phrack/56/ p56-0x05 (2000).

[Cowan 98] Cowan, C.; Pu, C.: Maier, D.; Hinton, H.; Walpole, J.; Bakke, P.; Beattie, S.; Grier, A.; Wagle, P.; & Zhang, Q. “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,” 63-77. *Proceedings of the Seventh USENIX Security Symposium*. San Antonio, TX, January 26-29, 1998. Berkeley, CA: USENIX Association, 1998.

[Cowan 00] Cowan, Crispin; Wagle, Perry; Pu, Calton; Beattie, Steve; & Walpole, Jonathan. “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade,” 119-129. *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX’00)*. Hilton Head Island, SC, January 25-27, 2000. Los Alamitos, CA: IEEE Computing Society, 2000.

[de Raadt 03] Theo de Raadt, Advances in OpenBSD, presented at CanSecWest, Vancouver, Canada. April 2003 http://www.openbsd.org/papers/csw03/mgp00001.html

57. http://www.research.avayalabs.com/project/libsafe

[Etoh 00] Etoh, Hiroaki & Yoda, K. *Protecting from stack-smashing attacks*[61], 2000.

[Jones 97] Jones, Richard W. M. & Kelley, Paul H. J. "Backwards-compatible bounds checking for arrays and pointers in C programs," 13-26. *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG'97)*. Linkoping, Sweden, May 26-27, 1997. Linkoping, Sweden: Linkopings Universitet, 1997.

[Ruwase 04] Ruwase, Olatunji & Lam, M. S. "A Practical Dynamic Buffer Overflow Detector," 159-169. *Proceedings of the 11th Annual Network and Distributed System Security Symposium*. San Diego, CA, February 5-6 2004. Reston, VA: Internet Society, 2004. http://suif.stanford.edu/papers/tunji04.pdf.

[Seacord 05] Robert C. Seacord. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005.

[Wagle 03] Wagle, Perry & Cowan, Crispin. "StackGuard: Simple Stack Smash Protection for GCC," 243-256. *Proceedings of the GCC Developers Summit*. Ottawa, Ontario, Canada, May 25-27, 2003.

[Wilander 03] Wilander, J. & Kamkar, M. "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," 149-162. *Proceedings of the 10th Network and Distributed System Security Symposium*. San Diego, California, February 6-7, 2003. Reston, VA: Internet Society, 2003. http://www.ida.liu.se/~johwi/ research_publications/paper_ndss2003_john_wilander.pdf.

Pearson Education, Inc. Copyright